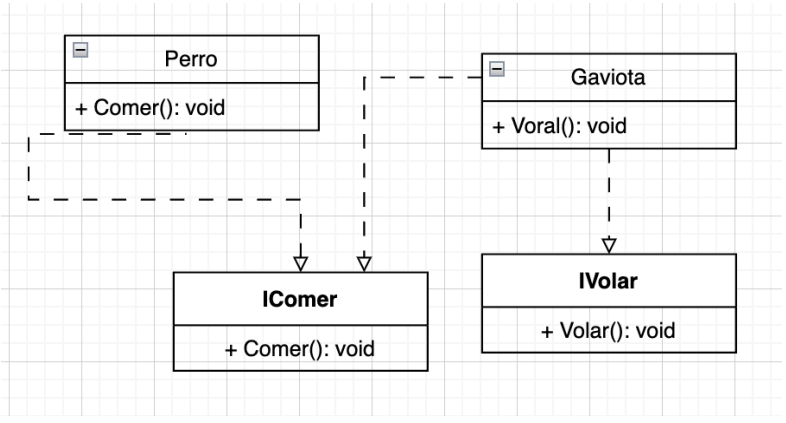
EJERCICIO 1

```
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 }
```

En este caso, se está violando el **Principio de sustitución de Liskov,** debido a que Perro no puede sustituir a la clase padre Animal, al no tener implementado el método Volar().

Para solucionarlo, se podría remover la clase abstracta Animal y crear dos interfaces IComer e IVolar, en las que estén los métodos que corresponden a cada una, para que luego la clase Perro implemente la interfaz IComer y defina la lógica del método Comer().

```
public interface IComer {
   void Comer();
}
public interface IVolar {
   void Volar();
}
public class Perro implements IComer {
   public void Comer() {
      //La logica de comer para el perro en especifico
   }
}
```



EJERCICIO 2

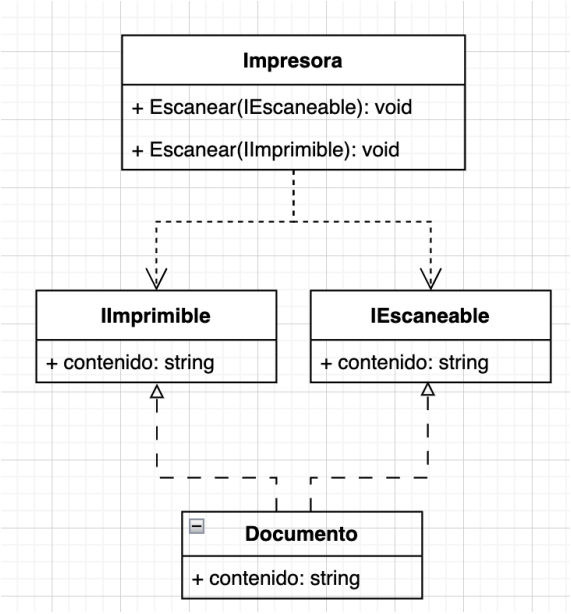
```
public class Documento {
  public string Contenido {
    get;
    set;
  }
}

public class Impresora {
  public void Imprimir(Documento documento) {
    Console.WriteLine(documento.Contenido);
  }

  public void Escanear(Documento documento) {
    // Código complejo para escaneo...
  }
}
```

En este ejercicio, también se está violando **DIP** debido a que la clase Impresora está dependiendo de una clase concreta, que es Documento, en lugar de depender de abstracciones, que es lo que se estipula en el principio nombrado.

```
public class Documento: IImprimible, IEscaneable {}
public interface IImprimible {
  string Contenido {
      get;
      set;
   }
}
public interface IEscaneable {
 string Contenido {
      get;
      set;
   }
}
public class Impresora {
   public void DocumentoEscaneado(IEscaneable documento) {
      // Código para escaneo
   }
   public void ImprimirDocumento(IImprimible documento) {
      Console.WriteLine(documento.Contenido);
   }
}
```



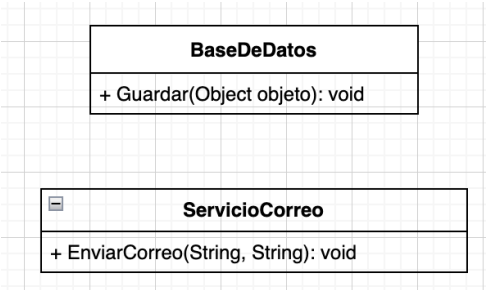
EJERCICIO 3

```
public class BaseDeDatos {
  public void Guardar(Object objeto) {
    // Guarda el objeto en la base de datos
  }

  public void EnviarCorreo(string correo, string
      mensaje) {
    // Envía un correo electrónico
  }
}
```

En este caso, se está violando el principio **SRP**, debido a que la clase BaseDeDatos tiene más de una razón para cambiar, las cuales podrían ser un cambio en el método Guardar o EnviarCorreo.

```
public class BaseDeDatos {
  public void Guardar(Object objeto) {
    // Guardar objeto en la base de datos
  }
}
public class ServicioCorreo {
  public void EnviarCorreo(String correo, String mensaje,) {
    // Enviar correo
  }
}
```



EJERCICIO 4

```
public class Robot {
  public void Cocinar() {
    // Cocina algo
  }

  public void Limpiar() {
    // Limpia algo
  }

  public void RecargarBateria() {
    // Recarga la batería
  }
}
```

En este caso, se está violando el principio **SRP**, debido a que la clase Robot tiene más de una razón para cambiar, las cuales podrían ser un cambio en los métodos Cocinar, Limpiar o RecargarBateria.

```
public interface ICocinar {
   public void cocinar();
}
public interface ILimpiar {
   public void limpiar();
}
public interface IRecargarBateria {
   public void recargarBateria();
}
public class Robot: ICocinar, ILimpiar, IRecargarBateria {
   public void cocinar() {
      Console.WriteLine("Cocinando...");
   }
   public void limpiar() {
      Console.WriteLine("Limpiando...");
   }
   public void recargarBateria() {
      Console.WriteLine("Recargando bateria...");
   }
}
```

EJERCICIO 5

```
public class Cliente {
  public void CrearPedido() {
    // Crear un pedido
  }
}
```

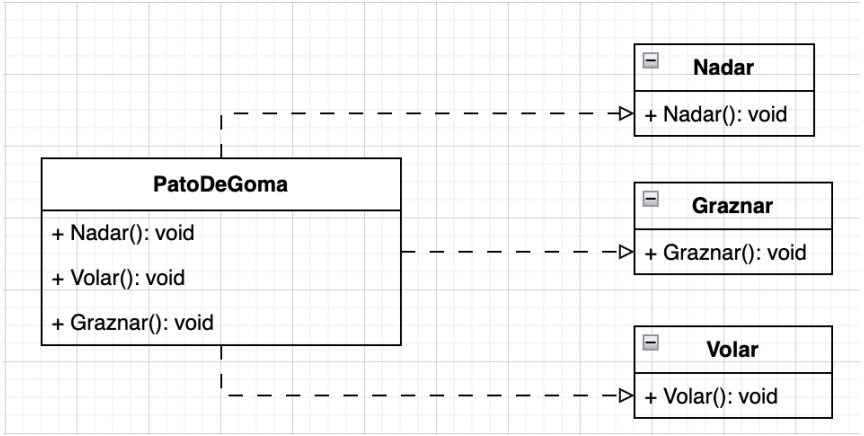
Esta clase está excelente. Sin embargo, el método CrearPedido() no está alineado con los otros métodos que podría tener la clase Cliente, este tendrá más de una razón de cambio. Incumpliendo **SRP**.

EJERCICIO 6

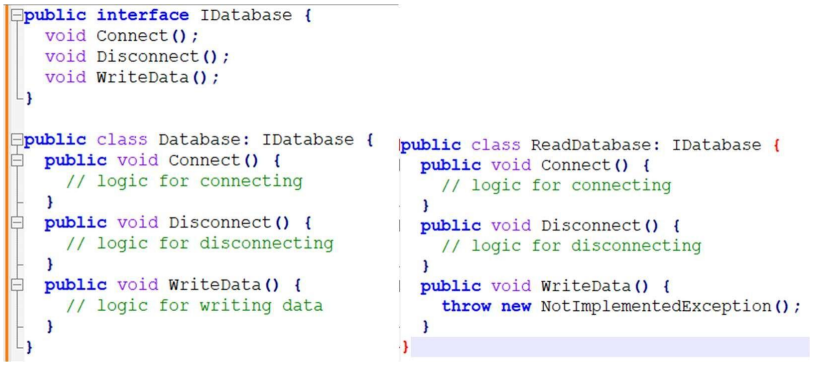
```
1  public class Pato {
2    public void Nadar() {
3      // Nada
4    }
5
6    public void Graznar() {
7      // Grazna
8    }
9
10   public void Volar() {
11     // Vuela
12   }
13 }
14
15 public class PatoDeGoma: Pato {
16   public override void Volar() {
17     throw new NotImplementedException();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

En este caso, se está violando el **Principio de sustitución de Liskov**, debido a que PatoDeGoma no puede sustituir a la clase padre Pato, al no tener implementado el método Volar().
Para solucionarlo, se podría remover la clase abstracta Pato y crear tres interfaces INadar, IGraznar e IVolar, en las que estén los métodos que corresponden a cada una, para que luego la clase PatoDeGoma implemente la interfaz INadar, IGraznar y defina la lógica de los métodos.

```
public interface INadar {
   public void nadar();
}
public interface IGraznar {
   public void graznar();
}
public interface IVolar {
   public void volar();
}
public class PatoDeGoma implements INadar, IGraznar{
   public void nadar() {
      System.out.println("Nadando");
   }
   public void graznar() {
      System.out.println("Graznando");
   }
}
```



EJERCICIO 7



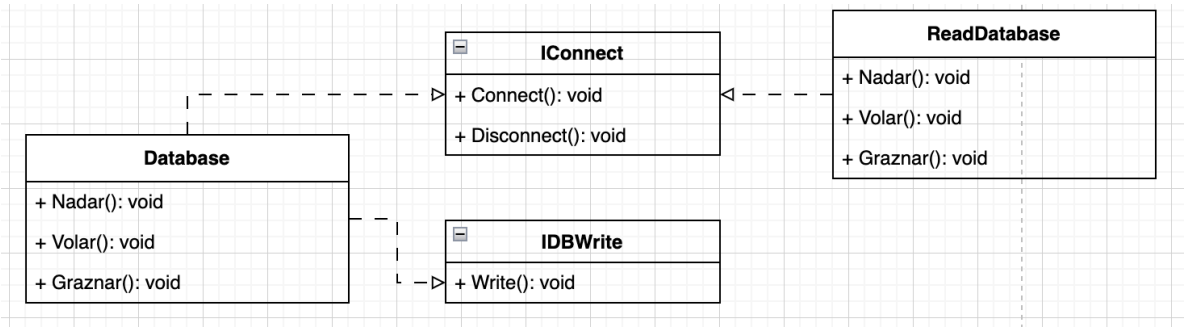
```
public interface IDatabase {
    void Connect();
    void Disconnect();
    void WriteData();
}

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();
    }
}
```

En el código dado, se está violando el principio **ISP** debido a que la clase ReadDatabase no define el método WriteData que se nombra en la interfaz que esta implementa, IDatabase.
Para solucionarlo, es mejor crear varias interfaces específicas en las que estén los métodos que se quieran definir al implementarlas: IConnect y IWriteData.
Luego, las clases implementarían solo las interfaces que le correspondan.

```
public interface IConnect {
    public void connect();
    public void disconnect();
}
public interface IWriteData {
    public void writeData(String data);
}
public class DataBase implements IConnect, IWriteData {
    @Override
    public void connect() {
        System.out.println("Connect");
    }
    @Override
    public void disconnect() {
        System.out.println("Disconnect");
    }
    @Override
    public void writeData(String data) {
        System.out.println("Write data");
    }
}
public class ReadDataBase implements IConnect {
    @Override
    public void connect() {
        System.out.println("Connect");
    }
    @Override
    public void disconnect() {
        System.out.println("Disconnect");
    }
}
```



EJERCICIO 8

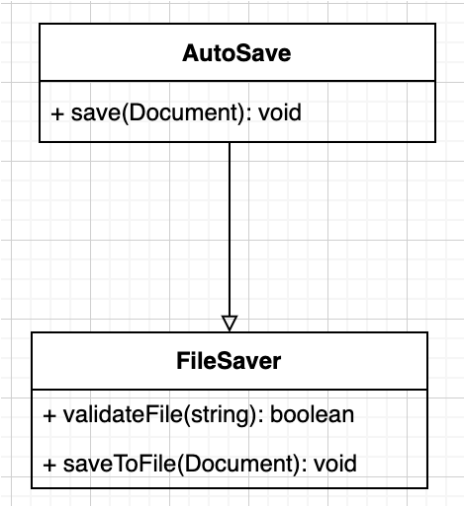
```
public class FileSaver {
    public void SaveToFile(string fileName, Document doc) {
        if (string.IsNullOrEmpty(fileName))
            throw new ArgumentException();
        // logic for saving the document
    }
}

public class AutoSave : FileSaver {
    public void Save(Document doc) {
        SaveToFile("", doc);
    }
}
```

SRP: El principio establece que una clase debe tener una única razón para cambiar. En el código proporcionado, la clase FileSaver tiene la responsabilidad de guardar un documento en un archivo, pero también realiza la validación del nombre del archivo. Podría ser más adecuado mover la validación del nombre del archivo a otra clase o método separado, lo que permitiría que FileSaver se centre únicamente en la tarea de guardar el documento.

```
public class FileSaver {
  private bool ValidateFileName(string filename) {
    return !string.IsNullOrEmpty(filename);
  }
  public void SaveToFile(string filename, Document doc) {
    if (!ValidateFileName(filename)) {
      throw new ArgumentException("Invalid filename");
    }
    // Lógica para guardar el documento en un archivo
  }
}

public class AutoSave: FileSaver {
  public void Save(Document doc) {
    SaveToFile("", doc);
  }
}
```



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; }
}
```

El **Principio de Responsabilidad Única** establece que una clase debe tener una única razón para cambiar. En el caso de la clase User, se le ha asignado la responsabilidad de determinar si un usuario puede editar una publicación. Sin embargo, también tiene una propiedad IsAdmin que indica si el usuario es administrador.

```
public class User {
   public bool IsAdmin { get; set; }
}
public class Post {
   public User Author { get; set; }
}
public class PostEditor {
   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
    }
}
```

La clase Music Player incumple con **OCP** porque si se desea agregar un nuevo archivo para reproducir, se debe modificar la clase MusicPlayer agregando otro método especial para reproducir este archivo.

```
public interface IMusicPlayer {
  void Play(string fileName);
}

public class Mp3Player implements IMusicPlayer {
  public void Play(string fileName) {
    // Lógica para reproducir archivos MP3
  }
}

public class WavPlayer implements IMusicPlayer {
  public void Play(string fileName) {
    // Lógica para reproducir archivos WAV
  }
}

public class FlacPlayer implements IMusicPlayer {
  public void Play(string fileName) {
    // Lógica para reproducir archivos FLAC
  }
}

public class PlayMusic {
  public void Play(IMusicPlayer musicPlayer) {
    musicPlayer.Play();
  }
}
```